Parallel Depth-First Search for Directed Acyclic Graphs

Maxim Naumov[1], Alysson Vrielink[1,2], and Michael Garland[1]

[1]NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050
[2]Stanford University, 2575 Sand Hill Road, Menlo Park, CA, 94025

Abstract

Depth-First Search (DFS) is a pervasive algorithm, often used as a building block for topological sort, connectivity and planarity testing, among many other applications. We propose a novel work-efficient parallel algorithm for the DFS traversal of directed acyclic graph (DAG). The algorithm traverses the entire DAG in a BFS-like fashion no more than three times. As a result it finds the DFS pre-order (discovery) and post-order (finish time) as well as the parent relationship associated with every node in a DAG. We analyse the runtime and work complexity of this novel parallel algorithm. Also, we show that unlike many of its predecessors, our algorithm is easy to implement and optimize for performance. In particular, we show that its CUDA implementation on the GPU outperforms sequential DFS on the CPU by up to $6\times$ in our experiments.

1 Introduction

Let a graph $G = (V, E)$ be defined by its vertex $V = \{1, ..., n\}$ and edge $E = \{(i_1, j_1), ..., (i_m, j_m)\}$ sets. Also, assume that it has n nodes and m edges.

The sequential depth-first search (DFS) algorithm was proposed in [25]. It is a pervasive algorithm, often used as a building block for topological sort [10, 18], connectivity and planarity testing [15, 28], among many other applications.

In the DFS traversal problem we are interested in finding parent, pre-order and post-order for every node in a graph. For example, for the graph on Fig. 1

NVIDIA Technical Report NVR-2017-001, March 2017.

these relationships are

$$\begin{aligned}
\text{node} &= \{a, b, c, d, e, f\} \\
\text{pre-order} &= \{0, 1, 4, 5, 2, 3\} \\
\text{post-order} &= \{5, 2, 4, 3, 1, 0\} \\
\text{parent} &= \{\emptyset, a, a, c, b, e\}
\end{aligned}$$

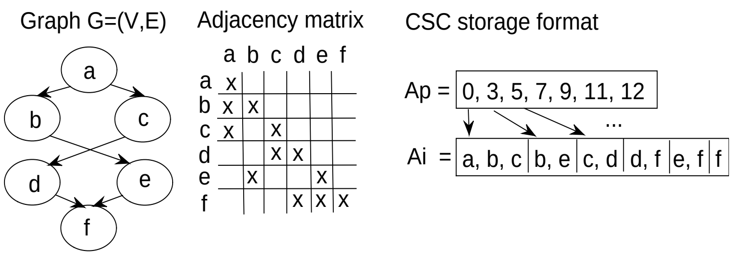
Graph G=(V,E)
Adjacency matrix
CSC storage format
a
b
c
d
e
f
a b c d e f
Ap = 0, 3, 5, 7, 9, 11, 12
...
Ai = a, b, c | b, e | c, d | d, f | e, f | f

Figure 1: Graph, adjacency matrix and CSC sparse storage format

Algorithm 1 Sequential DFS (Recursive)

```
 1: Let $G = (V, E)$ be a graph with a root node $r$.
 2: Let $d$ and $f$ be global variables initially set to 0.
 3: Let parent, pre- and post-order be global arrays.
 4: Call dfs(G,r) recursive routine defined below.
 5: routine dfs(G,p)
 6:     Mark node $p$ as visited
 7:     Set pre-order($p$) $= d$                          ▷ Discovery time
 8:     Set $d = d + 1$
 9:     Let $C_p$ be the set of children of $p$           ▷ Adjacent nodes
10:     for node $i \in C_p$ do                           ▷ Process in lexicographic order
11:         if $i$ has not been visited then
12:             parent($i$) $= p$
13:             Call dfs(G,$i$)
14:         end if
15:     end for
16:     Set post-order($p$) $= f$                         ▷ Finish time
17:     Set $f = f + 1$
18: end routine
```

The DFS traversal can be computed sequentially using many equivalent formulations of Alg. 1 [10]. Notice that unless stated otherwise, we assume that the children of a node are processed in some specified order, such as the lexicographic order, as shown on line 10 in Alg. 1.

2 Related Work

There has been a significant effort to create a parallel variation of DFS algorithm in the past. The DFS on planar graphs has been considered in [14, 23, 24]. In particular, it has been shown in [23] that for such graphs it is possible to perform a parallel DFS traversal in $\mathrm{O}(\log^2 n)$ time with n processors. On the other hand, the DFS on directed acyclic graphs (DAGs) has been considered in [13, 30], where it has been shown that for such graphs it is possible to find the DFS traversal in $\mathrm{O}(\log^2 n)$ time with $n^\omega/\log n$ processors, where $\omega < 2.373$ is the matrix multiplication exponent. Also, the DFS on directed graphs with cycles has been considered in [3, 4], where it has been shown that such traversal can be found in $\mathrm{O}(\sqrt{n}\log^{11} n)$ time with n^3 processors. Finally, a relaxation of the problem from lexicographic to unordered DFS, where the children of a node are not required to be explored in lexicographic order, but the parent relationship should still be such that it results in a DFS tree [17, 20, 22], has been explored for undirected graphs in [1, 2], where it has been shown that using randomized algorithms the unordered traversal can be obtained in $\mathrm{O}(\log^5 n)$ time with $n^{\omega+1}$ processors.

Recall that a given problem is in class $\mathcal{P}$ if there is a constant α such that this problem can be solved in $\mathrm{O}(n^\alpha)$ time on a single processor. Also, a given problem is in class $\mathcal{NC}$ and $\mathcal{RNC}$ if there are constants α and β such that this problem can be solved in $\mathrm{O}(\log^\alpha n)$ time on $\mathrm{O}(n^\beta)$ processors with deterministic and randomized algorithms, respectively [6]. It has been shown that the lexicographic DFS traversal problem for general graphs is $\mathcal{P}$-complete, in other words, all problems in class $\mathcal{P}$ can be reduced to it [21]. Therefore, a parallel algorithm that solves this problem in polylogarithmic time using polynomial number of processors would imply that $\mathcal{P} = \mathcal{NC}$ or $\mathcal{RNC}$, which is unlikely. However, it is misleading to use this result to simply state that DFS is an inherently sequential algorithm without specifying the taxonomy of the (lexicographic versus unordered) type of the traversal and the (planar, directed or general) class of graphs we are working with. As shown by the scientific literature review, we may conclude in particular that the lexicographic DFS traversal problem for planar graphs and DAGs $\in \mathcal{NC}$ and unordered DFS traversal problem for general graphs $\in \mathcal{RNC}$.

3 Contributions

Let us define a directed tree (DT) to be a DAG, where every node has a single parent. We will show that in a DT finding pre-order (discovery) and post-order (finish time) of a node is equivalent to computing an offset based on the number of nodes to the left and below yourself.

Also, we will propose two approaches for identifying the parent of a node corresponding to a lexicographic DFS of a DAG. The first will be based on the path comparisons and second will rely on the solution of a single source shortest path (SSSP) problem. We will use them to transform a DAG into a DT.

We will combine these results to develop a work-efficient parallel algorithm for computing lexicographic DFS traversal of a DAG. The algorithm will traverse the entire DAG in parallel in a Breadth-First Search (BFS)-like fashion no more than three times, with at least one traversal requiring that all edges to each parent are visited prior to proceeding to its children. As a result it will obtain the DFS parent, pre- and post-order relationship associated with every node.

We will show that for sparse graphs the parallel DFS can be performed in $O(\eta \log n)$ steps, where η is the length of the longest path between a root and a node. The length η depends on the connectivity structure of a graph. It can be short $O(1)$, balanced $O(\log n)$ or long $O(n)$ and we will propose several optimizations and a novel data structure that can effectively compress it.

Also, we will show that the work complexity of the proposed parallel DFS is $O(m+n)$, matching that of the sequential algorithm. We point out that to achieve this work complexity bound the number of processors $t \leq m+n$ actively doing work varies at each step of the algorithm.

Finally, we will show that unlike many of its predecessors, our algorithm is easy to implement and optimize for performance. In particular, we will show that its CUDA implementation on the GPU outperforms sequential DFS on the CPU by up to $6\times$ in our experiments.

4 Algorithm

Let us assume that we are working with a DAG, in other words, a graph $G = (V, E)$ that has directed edges and no cycles. The graph and its adjacency matrix can be stored in arbitrary data structures. Suppose that we use the standard CSR/CSC format, which simply concatenates all non-zero entries of the matrix in row/column-major order and records the starting position for the entries of each row/column, see Fig. 1. Notice that this format allows us to easily traverse and access information associated with outgoing edges.

Definition 1. *Let ς_p and ζ_p denote the number of nodes reachable under and including node p, where if a sub-graph is reachable from k multiple parents then its nodes are counted once and k times, respectively.*

Lemma 1. *For a DT, where each node has a single parent, $\zeta_p = \varsigma_p$ is simply the sub-graph size, while for a DAG we have $\zeta_p \geq \varsigma_p$.*

For example, in Fig. 1 we have $\zeta_a = 7$ and $\varsigma_a = 6$, because we double counted node f in the former case.

Also, notice that we have the recursive relationship

$$\zeta_p = 1 + \sum_{i \in C_p} \zeta_i \tag{1}$$

where C_p is a set of children of p. On the other hand, ς_p can be expensive to compute [5, 27].

Let C_p be an ordered set, then let

$$\widetilde{\zeta}_l = \sum_{i<l, i \in C_p} \zeta_i \tag{2}$$

Notice that $\widetilde{\zeta}_l$ and $\zeta_p - 1$ can be obtained as a result of the prefixsum of ζ_i starting with 0 for $i \in C_p$. Also, note that computation of $\widetilde{\zeta}_l$ does not include ζ_l because $i < l$ above.

The values ζ_p, ζ_i and $\widetilde{\zeta}_l$ for every node in the example on Fig. 1 are listed in Tab. 1. Notice that if we had laid out this data linearly we would have fit into CSR/CSC format. Thus, ζ_p can be stored and extracted in $O(1)$ time.

ζ_p	ζ_i for $i \in C_p$	$\widetilde{\zeta}_l$ = prefixsum(ζ_i,0)
$\zeta_a = 7$	[0,ζ_b,ζ_c]=[0,3,3]	[$\widetilde{\zeta}_b$,$\widetilde{\zeta}_c$,$(\zeta_a - 1)$]=[0,3,6]
$\zeta_b = 3$	[0,ζ_e] =[0,2]	[$\widetilde{\zeta}_e$,$(\zeta_b - 1)$] =[0,2]
$\zeta_c = 3$	[0,ζ_d] =[0,2]	[$\widetilde{\zeta}_d$,$(\zeta_c - 1)$] =[0,2]
$\zeta_d = 2$	[0,ζ_f] =[0,1]	[$\widetilde{\zeta}_f$,$(\zeta_d - 1)$] =[0,1]
$\zeta_e = 2$	[0,ζ_f] =[0,1]	[$\widetilde{\zeta}_f$,$(\zeta_e - 1)$] =[0,1]
$\zeta_f = 1$	[0] =[0]	[$(\zeta_f - 1)$] =[0]

Table 1: Values ζ_p, ζ_i and $\widetilde{\zeta}_l$ for every node on Fig. 1

Finally, using recursive relationship (1) the values ζ_p can be computed for a DT or DAG by traversing the graph bottom-up, from leafs to roots, as shown in Alg. 2.

Algorithm 2 Sub-Graph Size (bottom-up traversal)

```
1: Initialize all sub-graph sizes to 0.
2: Find leafs and insert them into queue $Q$.
3: while $Q \neq \{\emptyset\}$ do
4:    for node $i \in Q$ do in parallel
5:       Let $P_i$ be a set of parents of $i$ and queue $C = \{\emptyset\}$
6:       for node $p \in P_i$ do in parallel
7:          Mark $p$ outgoing edge $(p, i)$ as visited
8:          Insert $p$ into $C$ if all outgoing edges are visited
9:       end for
10:   end for
11:   for node $p \in C$ do in parallel
12:      Let $C_p$ be an ordered set of children of node $p$
13:      Compute a prefix-sum on $C_p$, obtaining $\zeta_p$
         (use lexicographic ordering of elements in $C_p$)
14:   end for
15:   Set queue $Q = C$ for the next iteration
16: end while
```

4.1 Directed Trees, Sub-Graph Size and DFS

Notice that for a node p in a DT the pre-order and post-order is based on the number of nodes visited before itself, which is related to the number of nodes found to the left of the node throughout different depth levels. In this section, we will use this observation to efficiently construct the pre-order and post-order based on the sub-graph size ζ_p. We note that a related but not the same approach has been used for finding Euler tours [7, 16, 26].

Definition 2. *Let a path from root r to node p be an ordered set of nodes $\mathfrak{P}_{r,p} = \{r, i_1, ..., i_{k-1}, p\}$, where k is the depth of the node p.*

Theorem 1. *Let ζ_i be the sub-graph size for node i in a DT and $\widetilde{\zeta}_l$ be the corresponding prefixsum value. Then,*

$$\textit{pre-order}(p) = k + \tau_p \tag{3}$$

$$\textit{post-order}(p) = (\zeta_p - 1) + \tau_p \tag{4}$$

where path $\mathfrak{P}_{r,p} = \{r, i_1, ..., i_{k-1}, p\}$ and

$$\tau_p = \sum_{l \in \mathfrak{P}_{r,p}} \widetilde{\zeta}_l \tag{5}$$

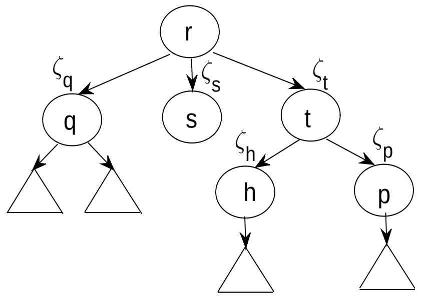

Figure 2: Sample DT

Proof. We will refer to the graph on Fig. 2, where triangles denote sub-graphs of arbitrary size, throughout the proof. Also, assume that lexicographic order is such that the node within the same level are processed left-to-right.

First, notice that in lexicographic DFS the pre- and post-order is not affected by the nodes to the right of the current node on the same level. Thus, t has no influence on node s.

Second, notice that in lexicographic DFS the nodes to the left of the current node on the same level, would have all been visited before the current node. Thus, q and s with their corresponding sub-graphs would have been visited before t.

Therefore, the number of nodes to the left of the node l on the same level is given by $\widetilde{\zeta}_l$, which was defined in (2) and can be computed using a prefixsum operation. In our example,

$$\begin{aligned} \widetilde{\zeta}_t &= \zeta_q + \zeta_s \\ \widetilde{\zeta}_p &= \zeta_h \end{aligned} \tag{6}$$

Third, notice that as we go down a level, all nodes on the previous level to the left of our parent node must have been visited before the current node. For instance, q and s must have been visited before node p.

Therefore, the number of nodes visited before node p at an arbitrary level is given by

$$\tau_p = \sum_{l \in \mathfrak{P}_{r,p}} \widetilde{\zeta}_l \tag{7}$$

Finally, notice that to compute the pre-order of the node we must include the nodes present directly on the path $\mathfrak{P}_{r,p}$ because they would have been discovered before the current node. If node p is at depth k, there are k such nodes.

Therefore,

$$\text{pre-order}(p) = k + \tau_p \tag{8}$$

On the other hand, to compute post-order of the node we must include the nodes under it because they would have been finished before the current node. Therefore,

$$\text{post-order}(p) = (\zeta_p - 1) + \tau_p \tag{9}$$

□

Also, notice that τ_p can be computed recursively using the following result.

Corollary 1. *Let a path from root r to node p be an ordered set of nodes $\mathfrak{P}_{r,i_k} = \{r, i_1, ..., i_{k-1}, p\}$. Then,*

$$\tau_p = \tau_{i_{k-1}} + \widetilde{\zeta}_p \tag{10}$$

Therefore, using recursive relationship (10) the pre- and post-order can be computed for a DT by traversing the graph top-down, from roots to leafs, and accumulating the sub-graph size as shown in Alg. 3.

Algorithm 3 Pre- and Post-Order (top-down traversal)

```
1: Initialize pre and post-order of every node to 0.
2: Find roots and insert them into queue $Q$.
3: while $Q \neq \{\emptyset\}$ do
4:     for node $p \in Q$ do in parallel
5:         Let pre = pre-order(p)
6:         Let post= post-order(p)
7:         Let $C_p$ be a set of children of $p$ and queue $P = \{\emptyset\}$
8:         for node $i \in C_p$ do in parallel
9:             Set pre-order($i$) = pre + $\widetilde{\zeta}_i$
10:            Set post-order($i$)= post+ $\widetilde{\zeta}_i$
11:            Mark $i$ incoming edge $(p, i)$ as visited
12:            Insert $i$ into $P$ if all incoming edges are visited
13:        end for
14:        Set pre-order($p$) = pre + depth($p$)
15:        Set post-order($p$)= post+ $\zeta_p$
16:    end for
17:    Set queue $Q = P$ for the next iteration
18: end while
```

So far we have shown how to compute a DFS traversal of a DT using Alg. 2 and 3. Next we will show how we can transform a DAG into a valid DT by selecting a single parent for every node, such that it corresponds to a DFS traversal. For this purpose we will develop Path- and SSSP-based complementary variations of the algorithm.

4.2 Path-based DFS

In the Path-based DFS we obtain a DT from a DAG by keeping track of multiple paths from root r to node p and selecting among them the DFS path according to the following rules.

Definition 3. *Let $\mathfrak{P}_{r,p} = \{r, i_1, i_2, ..., i_{k-1}, p\}$ and $\mathfrak{Q}_{r,p} = \{r, j_1, j_2, ..., j_{l-1}, p\}$ be two paths of potentially different length to node p. We say that path $\mathfrak{P}_{r,p}$ has the first lexicographically smallest node and denote it*

$$\mathfrak{P}_{r,p} < \mathfrak{Q}_{r,p} \tag{11}$$

when during the pair-wise comparison of the elements in the two paths going from left-to-right the path $\mathfrak{P}_{r,p}$ has the lexicographically smallest element in the first mismatch.

For example, in Fig. 1 the two paths to node f are

$$\mathfrak{P}_{a,f} = \quad [a, b, e, f] \tag{12}$$

$$\mathfrak{Q}_{a,f} = \quad [a, c, d, f] \tag{13}$$

When we compare these paths pairwise from left-to-right, we notice that the first mismatch between paths happens in the second pair $[\frac{b}{c}]$, where the lexicographically smallest digit b is contained in (12) and therefore we say that

$$\mathfrak{P}_{a,f} < \mathfrak{Q}_{a,f} \tag{14}$$

Notice that the next pairwise mismatch $[\frac{e}{d}]$, where the lexicographically smallest digit d is contained in (13), does not affect this decision.

Theorem 2. *Let $\mathfrak{P}_{r,p} = [r, i_1, i_2, ..., i_{k-1}, p]$ and $\mathfrak{Q}_{r,p} = [r, j_1, j_2, ..., j_{l-1}, p]$ be two paths to node p. If $\mathfrak{P}_{r,p} < \mathfrak{Q}_{r,p}$ then $\mathfrak{P}_{r,p}$ is the path taken by DFS.*

Proof. Let us prove the theorem by contradiction. Suppose $\mathfrak{P}_{r,p}$ has the first lexicographically smallest node left-to-right i at depth k, but path $\mathfrak{Q}_{r,p}$ is the one taken by DFS.

Notice that since i is the first lexicographically smallest node left-to-right, then all nodes preceding it in both paths must be the same. Also, notice that i must have been explored before the node at depth k in path $\mathfrak{Q}_{r,p}$ because it is lexicographically smallest. Finally, notice that DFS always explores the entire sub-graph under a node before proceeding. Therefore, DFS would have visited the entire sub-graph under i, including node p, before it was visited from path $\mathfrak{Q}_{r,p}$, which is a contradiction. □

Corollary 2. *Let $\mathfrak{S}$ be the set of all paths from root r to node p. The DFS traversal takes*

$$\mathfrak{P}_{r,p} = \min_{\mathfrak{Q}_{r,p} \in \mathfrak{S}} \mathfrak{Q}_{r,p} \tag{15}$$

Notice that the identification of the DFS path allows us to correctly select a single valid parent for each node. Finally, the algorithm that traverses the graph top-down, from roots to leafs, and transforms a DAG into a valid DT using path comparisons is shown in Alg. 4.

Algorithm 4 Compute DFS-Parent by Comparing Path (top-down traversal)

```
 1: Initialize path to $\{\emptyset\}$ and parent to $-1$ for every node.
 2: Find roots and insert them into queue $Q$.
 3: while $Q \neq \{\emptyset\}$ do
 4:    for node $p \in Q$ do in parallel
 5:       Let $C_p$ be a set of children of $p$ and queue $P = \{\emptyset\}$
 6:       for node $i \in C_p$ do in parallel
 7:          Let the existing path be $\mathfrak{Q}_{r,i}$
 8:          Let the new path be $\mathfrak{P}_{r,i}$
             ($\mathfrak{P}_{r,i}$ is a concatenation of path to $p$ & node $i$)
 9:          if $\mathfrak{P}_{r,i} \leq \mathfrak{Q}_{r,i}$ then
10:             Set $\mathfrak{Q}_{r,i} = \mathfrak{P}_{r,i}$
11:             Set parent($i$) $= p$
12:          end if
13:          Mark $i$ incoming edge $(p, i)$ as visited
14:          Insert $i$ into $P$ if all incoming edges are visited
15:       end for
16:    end for
17:    Set queue $Q = P$ for the next iteration
18: end while
```

In a naive implementation, to perform the comparison on line 9 in Alg. 4, we can store the path nodes in a linear array, align the arrays on the left and compare the elements pairwise left-to-right until a mismatch is found between the two arrays. In practice, the path length is between $\mathrm{O}(\log n)$ and $\mathrm{O}(n)$ and therefore, in order to shorten it, we perform the following optimizations to minimize the storage and comparison time requirements.

4.2.1 Path Static Pruning

Notice that when we look at two paths that reach the same node, there will be a parent p with outgoing edges (p, i) and (p, j) to nodes i and j, respectively, where one node will be preferred over the other due to its lexicographic ordering. It is the comparison of i and j stored in different paths in such a situation that allows us to distinguish between them. On the other hand, parent nodes p with a single outgoing edge (p, i) will never be a decision point on which we prefer one path over the other, because the path would have split before or after such point, it simply can not split at it.

This reasoning allows us to conclude the following theorem, which implies that we do not need to store nodes in the path whose parents have single outgoing edge.

Theorem 3. *If parent p has only a single outgoing edge (p, j) to node j then j does not need to be stored in the path, in other words, it will not affect the path comparison.*

For example, using static pruning we may conclude that the two paths to node f in Fig. 1 can be stored as

$$[a, b, f] \tag{16}$$

$$[a, c, f] \tag{17}$$

The effect of pruning the path is shown in Fig. 3, where we plot how much shorter the longest path became after pruning was applied to graphs in Tab. 3.

4.2.2 Path Compression

Notice that we can lexicographically associate an edge number with each outgoing edge from a node and store it instead of the node itself in the path. While storing a node typically requires 32 (or 64) bits, the number of outgoing edges is always less than or equal to the number of nodes, and in many cases significantly smaller. If there are at most μ outgoing edges from a node, than we only need $\log_2 \mu$ bits to store each element in the path.

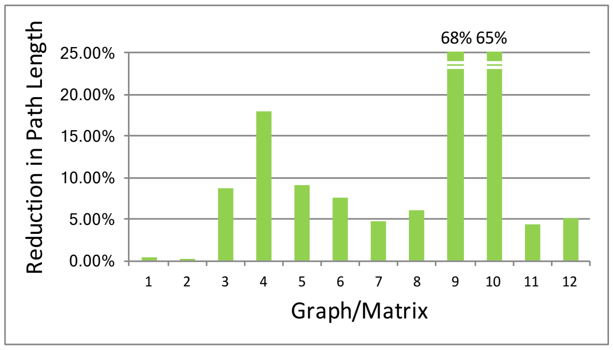

Figure 3: Reduction in longest path length with pruning

For example, $\mu = 2$ in the graph on Fig. 1 and therefore we only need a single bit to store the information for every node in the path. In fact, both paths to node f fit into 4 bits, and can be represented as

$$[0, 0, 0, 0] \tag{18}$$

$$[0, 1, 0, 0] \tag{19}$$

Let us define the compression rate as the ratio (# of bits required to store a node)/(# of bits required to store an edge). This compression rate is shown for realistic graphs from Tab. 3 on Fig. 4, where we have assumed that the numerator is 32 bits and therefore plot the ratio $32/\log_2 \mu$.

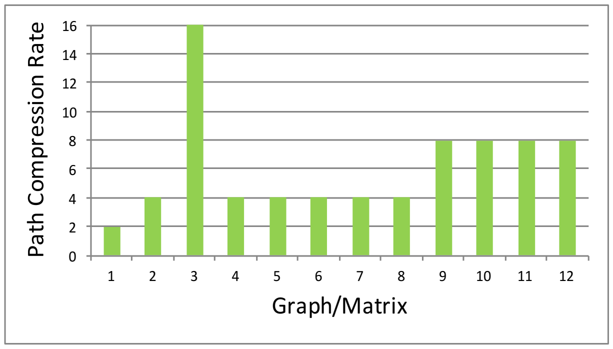

Figure 4: Path compression rate

4.2.3 Path Data Structure

Notice that there are two extremes for storing the path. On one hand, we could store the path elements in a linear array. In this case comparisons would be easily performed in parallel, but the setting of a new path would require a copy of data. Also, this approach is relatively expensive in terms of memory. On the other hand, we could perform path comparisons by simply traversing the parent pointers back to the root. In this case setting the path would be easy, but the comparisons would be relatively expensive and would need to be done sequentially.

We propose a special data structure for storing the path that combines the two approaches and can be easily tailored for the hardware architecture at hand. It resembles a linked list of blocks, where each block stores part of the path.

First, we let the user specify the size of the block of elements of the path to be stored linearly, which for example could be 32 elements corresponding to a warp (32 threads) on the GPU or the width of a SIMD unit on the CPU. We reserve two elements of this blocks to indicate the tail size and the "parent" relationship explained next.

Then, we chain these blocks to create the full path in the following fashion. If the path fits into its fixed block than we store its size in the first reserved element and we are done. If it does not fit, we store a tail of the path in its fixed block and use the second reserved element to point to its "parent" block, where the beginning of the path can be found. We apply these rules recursively.

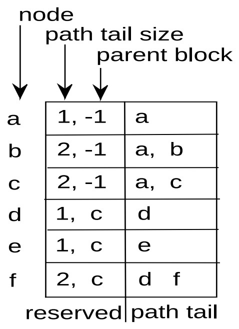

Figure 5: Path data structure

For example, suppose the size of the block is 2, then the path data structure for graph on Fig. 1 is shown in Fig. 5, where -1 indicates no "parent".

Notice that by chaining the blocks in this way we reuse the path already stored for earlier nodes therefore saving memory. Also, we allow the comparisons to be performed in parallel and enable early exit if the same "parent" is detected for both paths. Moreover, the cost of setting a new path is proportional to its tail size only.

Finally, the parallel DFS traversal of a DAG can be computed as shown in Alg. 5, where the first phase often consumes more than 80% of the total time as shown in the profiling of realistic examples from Tab. 3 on Fig. 6.

Algorithm 5 Parallel DFS (Path)

```
1: Let graph $G = (V, E)$ and its adjacency matrix $A$
2: Run Alg. 4                    ▷ Phase 1: Transform DAG to DT
3: Run Alg. 2                    ▷ Phase 2: Compute sub-graph size
4: Run Alg. 3                    ▷ Phase 3: Compute pre- and post-order
5: Resulting in parent, pre- and post-order for every node.
```

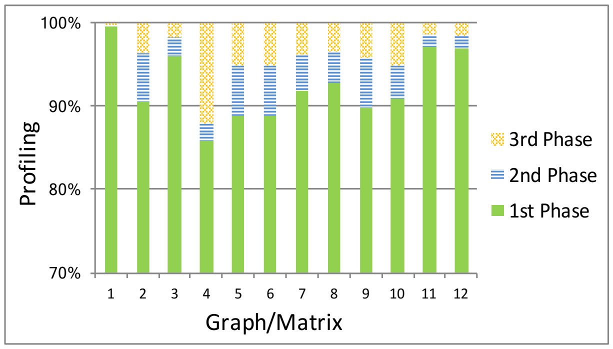

Figure 6: Profiling of time consumed by three phases of Path-based DFS

4.3 SSSP-based DFS

In the Single-Source Shortest Path (SSSP)-based DFS we construct a DT from a DAG implicitly. We do that by associating a positive weight with each edge in the DAG, such that the shortest path selected during the solution of the SSSP problem on this weighted DAG results precisely in the path selected by DFS.

Similarly to (2), let us define

$$\bar{\zeta}_l = 1 + \sum_{i<l, i \in C_p} \zeta_i = 1 + \widetilde{\zeta}_l \tag{20}$$

where C_p is a ordered set of children of p. Notice that $\bar{\zeta}_l$ and ζ_p can be obtained as a result of the prefixsum of ζ_i starting with 1 (rather than 0) for $i \in C_p$.

Notice that ζ_p and $\bar{\zeta}_l$ can be computed for a DAG using Alg. 2, because it propagates ζ_i to all parents by default. For example, the values ζ_p, ζ_i and $\bar{\zeta}_l$ for every node in a graph on Fig. 1 are listed in Tab. 2.

ζ_p	ζ_i for $i \in C_p$	$\bar{\zeta}_l$ = prefixsum(ζ_i,1)
$\zeta_a = 7$	[1,ζ_b,ζ_c]=[1,3,3]	[$\bar{\zeta}_b$,$\bar{\zeta}_c$,ζ_a]=[1,4,7]
$\zeta_b = 3$	[1,ζ_e] =[1,2]	[$\bar{\zeta}_e$,ζ_b] =[1,3]
$\zeta_c = 3$	[1,ζ_d] =[1,2]	[$\bar{\zeta}_d$,ζ_c] =[1,3]
$\zeta_d = 2$	[1,ζ_f] =[1,1]	[$\bar{\zeta}_f$,ζ_d] =[1,2]
$\zeta_e = 2$	[1,ζ_f] =[1,1]	[$\bar{\zeta}_f$,ζ_e] =[1,2]
$\zeta_f = 1$	[1] =[1]	[ζ_f] =[1]

Table 2: Values ζ_p, ζ_i and $\bar{\zeta}_l$ for every node on Fig. 1

Now, let us state the following key result that relates DFS traversal to the SSSP problem.

Theorem 4. *Let $\bar{\zeta}_l$ be the weight associated with every node l in a DAG. Suppose that the shortest path between root r and node p based on these weights is $\mathfrak{P}_{r,p}$. Then, $\mathfrak{P}_{r,p}$ is also the path taken by DFS.*

Proof. Let the weight $\omega_{\mathfrak{P}_{r,p}}$ of the path $\mathfrak{P}_{r,p}$ be the sum of the weight of all nodes in it

$$\omega_{\mathfrak{P}_{r,p}} = \sum_{l \in \mathfrak{P}_{r,p}} \bar{\zeta}_l \tag{21}$$

Notice that $\bar{\zeta}_l \geq 1$, therefore for non-empty path $\omega_{\mathfrak{P}_{r,p}} \geq 1$.

Further, notice that using (1) and (20), we may write

$$\begin{aligned} \zeta_p &= 1 + \sum_{i \in C_p} \zeta_i \\ &= 1 + \sum_{i<l, i \in C_p} \zeta_i + \zeta_l + \sum_{i>l, i \in C_p} \zeta_i \\ &= \bar{\zeta}_l + \zeta_l + \sum_{i>l, i \in C_p} \zeta_i \end{aligned} \tag{22}$$

for some node l among the children of p.

Let there be two distinct path $\mathfrak{P}_{r,k} = \{r, ..., k\}$ and $\mathfrak{Q}_{r,k} = \{r, ..., k\}$ from root r to node k. Let $[\frac{i}{j}]$ be the first mismatch in left-to-right pairwise comparison of these paths, with $i \in \mathfrak{P}_{r,k}$ being lexicographically smaller than $j \in \mathfrak{Q}_{r,k}$. Also,

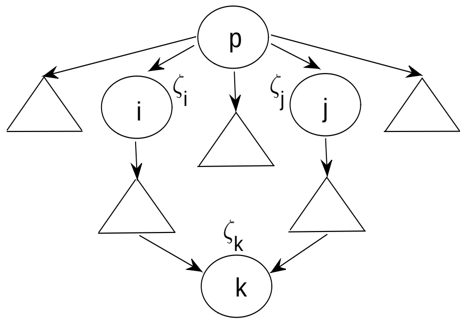

Figure 7: Two distinct paths $\mathfrak{P}_{p,k}$ and $\mathfrak{Q}_{p,k}$

let their parent p belong to both paths $\mathfrak{P}_{r,k}$ and $\mathfrak{Q}_{r,k}$. Notice that node p can be the root r, while i and j can be the node k, as shown in Fig 7.

Notice that based on the above assumptions paths are the same from root r to node p, therefore we only need to consider the difference in path weight starting at node p.

Then, notice that the weight of path $\mathfrak{Q}_{p,k}$ is given by

$$\begin{aligned}
\omega_{\mathfrak{Q}_{p,k}} &= \sum_{l \in \mathfrak{Q}_{p,k}} \bar{\zeta}_l \\
&\geq \bar{\zeta}_j \geq \bar{\zeta}_i + \zeta_i && (23)\\
&\geq \bar{\zeta}_i + \sum_{z \in \mathfrak{P}_{i,k}} \bar{\zeta}_z + \zeta_k && (24)\\
&= \sum_{z \in \mathfrak{P}_{p,k}} \bar{\zeta}_z + \zeta_k && (25)\\
&> \sum_{z \in \mathfrak{P}_{p,k}} \bar{\zeta}_z = \omega_{\mathfrak{P}_{p,k}} && (26)
\end{aligned}$$

where we have used (20) to obtain (23), applied (22) recursively, while dropping the third term, to obtain (24), regrouped the terms to obtain (25) and took advantage of the fact that $\zeta_k \geq 1$ to obtain (26).

Thus, we have proven that the shortest path computed using weights $\bar{\zeta}_l$ indeed coincides with the lexicographically smallest DFS path. $\square$

4.3.1 SSSP Algorithms

There exist a variety of algorithms for computing SSSP [10, 11]. We propose a specific approach for a DAG that traverses the graph top-down, from roots to leafs, in Alg. 6. In our experiments, this custom approach often outperforms the parallel SSSP implemented in GUNROCK [11, 29], as shown in Fig. 8.

We point out that using the custom implementation is particularly advantageous because we have access to the queue of nodes that belong to each level of the graph from the previous phase of the algorithm. This additional information allows for a significant speedup in the custom implementation.

Algorithm 6 Compute DFS-Parent by Tracking SSSP (top-down traversal)

```
1: Initialize cost to $\infty$ and parent to $-1$ for every node.
2: Find roots, set cost to 0 and insert them into queue $Q$.
3: while $Q \neq \{\emptyset\}$ do
4:     for node $p \in Q$ do in parallel
5:         Let $C_p$ be a set of children of $p$ and queue $P = \{\emptyset\}$
6:         for node $i \in C_p$ do in parallel
7:             Let current cost for node $i$ be stored in cost$(i)$
8:             Let new cost $\alpha$ = cost$(p)$ + $\bar{\zeta}_i$
9:             if $\alpha <$ cost$(i)$ then
10:                Set cost$(i) = \alpha$
11:                Set parent$(i) = p$
12:            end if
13:            Mark $i$ incoming edge $(p, i)$ as visited
14:            Insert $i$ into $P$ if all incoming edges are visited
15:        end for
16:    end for
17:    Set queue $Q = P$ for the next iteration
18: end while
```

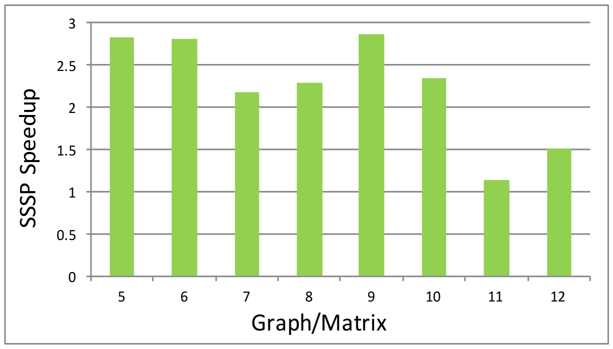

Figure 8: Speedup of custom vs. GUNROCK SSSP implementation

4.3.2 Pre- and Post-order

Once we have resolved the parent relationship, we could simply apply Alg. 2 and 3 as done in phases 2 and 3 of Path-based approach to find the pre- and post-order traversal of every node. However, in this section we will show that it is possible to completely avoid those traversals when pre-order only is needed.

Corollary 3. *The ordering of nodes based on their path weight $\omega_{\mathfrak{P}_{r,p}}$ or pre-order time is the same.*

Proof. First, notice that we can not directly use Thm. 1 to compute pre-order and post-order of the nodes because the weights $\bar{\zeta}_l = 1 + \widetilde{\zeta}_l$ have been computed for a DAG and not a DT in this section.

However, notice that computed path weight $\omega_{\mathfrak{P}_{r,p}}$ imposes an ordering on the nodes. For instance, nodes in the same path must come one after another according to their path weight. Also, nodes with the same parent are ordered lexicographically as shown in Thm. 4. If we apply this arguments recursively, we may conclude that the ordering based on path weight $\omega_{\mathfrak{P}_{r,p}}$ coincides with pre-order ordering. □

On the other hand, notice that post-order can be expressed in terms of pre-order of a node as shown in the next Corollary. Therefore, if depth k is known for every node then we can completely avoid calling both Alg. 2 and 3 for computation of the pre- and post-order traversal. If the depth is not available then we can call Alg. 2 to obtain and use it for computing the post-order only.

Corollary 4. *For a DT the post-order of node p can be expressed through its pre-order*

$$\textit{post-order}(p) = \textit{pre-order}(p) - k + (\zeta_p - 1) \tag{27}$$

where k is the depth and ζ_p is the sub-graph size.

Proof. Follows directly from Thm. 1. □

Finally, the SSSP-based parallel DFS traversal of a DAG can be computed as shown in Alg. 7. Notice that for graphs listed in Tab. 3 its first and second phase often consume more than 80% of the total time as shown in Fig. 9.

Also, notice that the time consumed by the first and last phases using Alg. 2 for DAG and DT traversals, respectively, is starkly different. This discrepancy is due mainly to two factors: (i) in the former case the graph has many more edges, and (ii) in the latter case a node may be scheduled into the queue earlier because its dependencies, have been already satisfied. It is important to keep this distinction in mind when analysing the algorithm.

Algorithm 7 Parallel DFS (SSSP)

```
1: Let graph $G = (V, E)$ and its adjacency matrix $A$
2: Run Alg. 2                                  ▷ Phase 1: Compute DAG Edge Weights
3: Run Alg. 6                                  ▷ Phase 2: Transform DAG to DT
4: Sort nodes based on SSSP length             ▷ Phase 3a: Pre-order
5: if post-order needed then
6:     Run Alg. 2 and use (27)                 ▷ Phase 3b:Post-order
7: end if
8: Resulting in parent, pre- and post-order for every node.
```

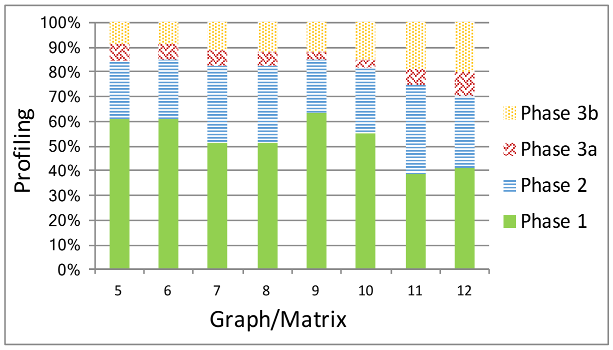

Figure 9: Profiling of time consumed by three phases of SSSP-based DFS

4.3.3 Weight Overflow

A potential disadvantage of the SSSP-based algorithm is that the weights $\bar{\zeta}_l$ computed by the first phase of the algorithm might overflow. Notice that in Alg. 2 for DAGs at every depth of the graph we are accumulating $\bar{\zeta}_l$, rather than the sub-graph size ς_l, which double counts the node with multiple parents. It is possible to come up with adversarial patterns for the algorithm, such that it will double the weight at every depth. Therefore, if the weight is stored in 64 bit integer then overflow could happen at depth 64. We have found such patterns to be unlikely in realistic scenarios, but there are situations where we still need a significant number of bits to represent the weights. For example, in Tab. 3 the matrix `delaunay_n21` requires 1024 bits and matrix `auto` requires 512 to store their weights. In practice, it is possible to handle such overflow by using arbitrary precision arithmetic, but this was beyond the scope of this study and we have left it as a future work.

4.4 Complexity Analysis

Let n and m be the number of nodes and edges in a DAG. Also, let us assume a standard theoretical PRAM (CREW) model for the formal analysis [16].

Definition 4. *Let k_i be the number of elements inserted into the queue at iteration i and $k = \max_i k_i$ in Alg. 2-4 and 6.*

Definition 5. *Let d_i denote the degree of node i and $d = \max_i d_i$ denote the maximum degree in a DAG.*

Definition 6. *Let δ_i denote the minimum depth*[1] *of node i and $\delta = \max_i \delta_i$ denote the diameter in a DAG.*

Definition 7. *Let η_i denote the maximum depth*[2] *of node i and $\eta = \max_i \eta_i$ denote the length of the longest path in a DAG.*

Lemma 2. *The parallel prefixsum of n numbers, can be computed in $O(\log n)$ steps. Also, the algorithm performs $O(n)$ work.*

Proof. Please refer to [8]. $\square$

Lemma 3. *The parallel (comparison-based) sort of n numbers, can be computed in $O(\log n)$ steps. The algorithm performs $O(n \log n)$ work.*

Proof. Please refer to [9]. $\square$

Lemma 4. *Let $n = \min(n_1, n_2)$ then identifying the first left-to-right pair of digits that is different in two sequences of n_1 and n_2 numbers can be performed in $O(\log n)$ steps. Also, the algorithm performs $O(n)$ work.*

Proof. Let the sequences be aligned on the leftmost digit and let processor i be assigned the i-th pair of numbers. Let it compute either index i if its pair of digits is different or ∞ otherwise, which can be done in $O(1)$. Then, we can find a minimum of the results using a binary tree-like reduction implemented using a prefixsum, see Lemma 2. $\square$

Lemma 5. *The queue can be implemented such that parallel insertion and extraction of n numbers, can be performed in $O(\log n)$ and $O(1)$ steps, respectively. Also, the algorithm performs $O(n)$ work.*

[1]The length of the shortest path from the root to a node.

[2]The length of the longest path from the root to a node.

Proof. Suppose that we store the queue data in linear array with the start and end indices. Also, suppose that at each iteration we extract all data elements between the start and end indices and we insert new data elements immediately after the end index. Then, we adjust the start and end indices to point to the newly inserted data for the next iteration and repeat the process.

In this scenario we can extract all data elements in parallel in O(1) steps. In order to insert the elements in parallel without write conflicts, we may choose to assign a unique write index to each processor to indicate a particular location in the array where the element can be written safely. Notice that such a write index can be computed using a prefixsum, see Lemma 2. □

Notice that for DT $\eta = \delta$, while for a DAG $\eta \geq \delta$. Also, notice that $k, d, \delta, \eta \leq n$ for a DAG. Let us now state the complexity of each phase of parallel DFS.

Theorem 5. *Alg. 2 takes O($\eta(\log d + \log k)$) steps and performs O($m+n$) total work to traverse a DAG. The number of processors $t \leq m + n$ actively doing work varies at each step of the algorithm.*

Proof. The Alg. 2 performs η while loop iterations because we enqueue a node once all its edges are visited.

In each iteration, notice that we can access $\zeta_{p,i}$ on line 7 in O(1). Also, the parallel prefixsum performed on line 14 takes no more than O($\log d$), see Lemma 2. We might need to perform multiple of these prefixsums at once, but we will never be using more than m edges. Therefore, we need at most O(m) processors for it. Also, at each iteration we might need to insert $k \leq n$ elements into a queue on line 9, which can be done in O($\log k$), see Lemma 5.

Notice that work performed for each iteration is O($d_i + k_i$). Since $m = \sum_i d_i$ and $n = \sum_i k_i$, the total work performed is O($m + n$). □

Theorem 6. *Alg. 3 takes O($\eta \log k$) steps and performs O(n) total work to traverse a DAG. The number of processors $t \leq n$ actively doing work varies at each step of the algorithm.*

Proof. The Alg. 3 performs O(η) while loop iterations, with each iteration requiring O(1) operations. However, at each iteration we might need to insert $k \leq n$ elements into a queue on line 12, which can be done in O($\log k$), see Lemma 5. The work performed for each iteration is O(k_i). Since $n = \sum_i k_i$, the total work performed is O(n). □

Theorem 7. *Alg. 4 takes O($\eta(\log \eta + \log k)$) steps and performs O($\eta m + n$) total work to traverse a DAG. The number of processors $t \leq \eta d + n$ actively doing work varies at each step of the algorithm.*

Proof. The Alg. 4 performs $O(\eta)$ while loop iterations.

Notice that per iteration the most time consuming part of the algorithm involves path comparison $\mathfrak{P}_{r,i} \leq \mathfrak{Q}_{r,i}$ on line 9. It can be performed in $O(\log \eta)$, see Lemma 4. We might need to perform multiple path comparisons at once, therefore we might need $O(\eta d)$ processors. The work performed for each path (sequence) comparison is $O(\eta)$. There are no more than $m = \sum_i d_i$ sequences to compare, therefore we perform $O(\eta m)$ total work for it.

The path exchange $\mathfrak{Q}_{r,i} = \mathfrak{P}_{r,i}$ on line 10 can be performed in $O(1)$, because we only replace the (fixed size) path tail, using previously discussed data structure, see Fig. 5.

Also, at each iteration we might need to insert $k \leq n$ elements into a queue on line 14, which can be done in $O(\log k)$, see Lemma 5. The work performed for this operation at each iteration is $O(k_i)$. Since $n = \sum_i k_i$, the total work performed for this operation is $O(n)$. □

Theorem 8. *Alg. 6 takes $O(\eta \log k)$ steps and performs $O(n)$ total work to traverse a DAG. The number of processors $t \leq n$ actively doing work varies at each step of the algorithm.*

Proof. Follows proof of Thm. 6. □

The complexity of sequential DFS is $O(m + n)$. The complexity of the novel parallel DFS variants is stated below.

Corollary 5. *The Path-based DFS in Alg. 5 takes $O(\eta(\log d + \log k + \log \eta))$ steps and performs $O(m + n + \eta m)$ total work to traverse a DAG. The number of processors $t \leq m + n + \eta d$ actively doing work varies at each step of the algorithm.*

Notice that in practice $\eta m \rightarrow m$ because the data structure for storing the path detects the same "parent" block during comparisons, which often implicitly eliminates additional work, see Fig. 5. Also, recall that $d, k, \eta \leq n$ in a DAG and therefore the Path-based algorithm takes no more than $O(\eta \log n)$ steps and performs $O(m + n)$ total work.

Corollary 6. *The SSSP-based DFS in Alg. 7 takes $O(\eta(\log d + \log k))$ steps and performs $O(m + n + n \log n)$ total work to traverse a DAG. The number of processors $t \leq m + n$ actively doing work varies at each step of the algorithm.*

Notice that the work complexity of parallel sort can be improved by using non-comparison based algorithms, such as radix sort, in which case $n \log n \rightarrow n$. Since $d, k \leq n$ in a DAG, we may conclude that SSSP-based algorithm takes no more than $O(\eta \log n)$ steps and performs $O(m + n)$ total work.

Moreover, real-world graphs are often sparse with the number of edges $m = \gamma n$ where γ is some small constant. Consequently, in practice the total work performed for both Path- and SSSP-based parallel DFS is often $\mathrm{O}(n)$.

Finally, notice that while the parallel complexity of BFS is related to diameter δ [19], we have shown that the parallel complexity of lexicographic DFS is related to the length of the longest path η in a DAG.

5 Experiments

We study the performance of the DFS algorithm on a variety of graphs from the UFSMC and DIMACS collections [12] shown in Tab. 3. These graphs are selected from different applications and have very different node degree, longest path length and other characteristics, which allows us to make a more substantive analysis of the performance. When necessary we create DAGs based on these general graphs by dropping the back edges, in other words, only considering the lower triangular part of the adjacency matrix.

#	Graph	n	m	Application
1.	coPapersDBLP	540487	15251812	Citations
2.	auto	448696	3350678	Numeric. Sim.
3.	hugebubbles-000	18318144	30144175	Numeric. Sim.
4.	delaunay_n24	16777217	52556391	Random Tri.
5.	il2010	451555	1166978	Census Data
6.	fl2010	484482	1270757	Census Data
7.	ca2010	710146	1880571	Census Data
8.	tx2010	914232	2403504	Census Data
9.	great-britain_osm	7733823	8523976	Road Network
10.	germany_osm	11548846	12793527	Road Network
11.	road_central	14081817	21414269	Road Network
12.	road_usa	23947348	35246600	Road Network

Table 3: Sample DIMACS graphs/adjacency matrices

The experiments are performed on the workstation with Intel Core i7-3930K @3.2GHz CPU and Nvidia Pascal TitanX GPU with Ubuntu 14.04 LTS OS and CUDA Toolkit 8.0. We use standard sequential implementation of the DFS [10], written in C programming language and compiled with gcc compiler and -O3 optimization flags. We compare it against the Path- and SSSP-based parallel DFS implementations written in CUDA and compiled with nvcc compiler with -O3 optimization flags.

The comparison of sequential and parallel DFS is shown on Fig. 10. The performance of the algorithm depends highly on the sparsity pattern of the adjacency matrix, which reflects the connectivity of the DAG. On one hand, for graphs with high node degree, such as social network `coPapersDBLP`, the Path-based algorithm does not perform well because the length of the longest path η is very large. On the other hand, for graphs with low node degree, such as road network `road_central`, the algorithm performs exceedingly well. In this case the path is often short and can be compressed further using optimizations.

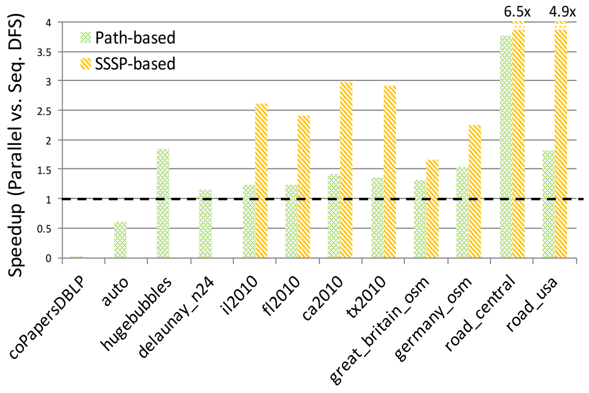

Figure 10: Speedup of Parallel Path- and SSSP-based vs. Sequential DFS

	DFS	Path-based Parallel DFS			
#	Seq.	Phase 1	Phase 2	Phase 3	Total
1.	0.063	5.112	0.0102	0.015	5.138
2.	0.033	0.045	0.0029	0.001	0.050
3.	0.748	0.386	0.0090	0.007	0.402
4.	0.781	0.576	0.0135	0.081	0.671
5.	0.016	0.011	0.0008	0.001	0.013
6.	0.018	0.013	0.0009	0.001	0.014
7.	0.029	0.019	0.0009	0.001	0.020
8.	0.033	0.023	0.0009	0.001	0.025
9.	0.260	0.179	0.0121	0.008	0.199
10.	0.446	0.260	0.0113	0.014	0.286
11.	1.045	0.267	0.0041	0.003	0.275
12.	0.791	0.418	0.0066	0.006	0.431

Table 4: Time (s) for seq. and Path-based parallel DFS

	DFS	SSSP-based Parallel DFS				
#	Seq.	Phase 1	Phase 2	Phase 3a	Phase 3b	Total
5.	0.016	0.004	0.001	0.000	0.001	0.006
6.	0.018	0.005	0.002	0.000	0.001	0.007
7.	0.029	0.005	0.003	0.001	0.001	0.009
8.	0.033	0.006	0.004	0.001	0.001	0.011
9.	0.260	0.098	0.034	0.005	0.018	0.154
10.	0.446	0.115	0.053	0.007	0.028	0.203
11	1.045	0.061	0.058	0.010	0.030	0.158
12.	0.791	0.066	0.047	0.015	0.032	0.160

Table 5: Time (s) for seq. and SSSP-based parallel DFS

Also, notice that when there is no weight overflow the SSSP-based DFS attains higher performance than Path-based DFS. In a way it exchanges the path comparisons for weight additions, which are more computationally efficient. However, in practice the algorithms are complementary. The former works best when there is limited nesting of the sub-graphs (so that the computation of the weight does not overflow and require special handling), while the latter works best when the path to every node is relatively short (so that path comparison can be done quickly).

Finally, the detailed results are stated in Tab. 4 and 5, where results for which 32 bit integer weights overflowed are not present.

6 Conclusion

In this paper we have developed a work-efficient parallel DFS algorithm that finds parent, pre- and post-order relationship for every node in a DAG. The algorithm performs up to three BFS-like traversals of the DAG and has Path- and SSSP-based complementary variations. We have proven that both variations of the algorithm obtain correct results and performed their runtime and work analysis.

The performance of the parallel algorithm depends highly on the connectivity of the DAG, in other words, the sparsity pattern of the adjacency matrix. In our experiments its parallel implementation performed particularly well on DAGs with low node degree, such as census and road networks, where it has outperformed by up to 6× the sequential DFS.

7 Acknowledgments

The authors would like to thank Isaac Gelado for providing a locking library on the GPU that was used during path comparisons and exchanges, as well as Steven Dalton, Alex Fit-Florea, Duane Merrill and Sean Treichler for insightful comments and suggestions.

References

[1] U. A. Acar, A. Chargueraud, and M. Rainey. A work-efficient algorithm for parallel unordered depth-first search. *Proc. Supercomputing*, 67, 2015.

[2] A. Aggarwal and R. J. Anderson. A random NC algorithm depth first search. *Proc. ACM Symp. Theory of Computing*, pages 325–334, 1987.

[3] A. Aggarwal, R. J. Anderson, and M. Y. Kao. Parallel depth-first search in directed graphs. *Proc. ACM Symp. Theory of Computing*, pages 297–308, 1989.

[4] A. Aggarwal, R. J. Anderson, and M. Y. Kao. Parallel depth-first search in general directed graphs. *SIAM Journal on Computing*, 19:397–409, 1990.

[5] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37:441–456, 2004.

[6] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, New York, NY, 2009.

[7] M. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[8] G. E. Blelloch. Prefix sums and their applications. *Carnegie Mellon University*, CS-TR-190, 1990.

[9] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17:770–785, 1988.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Camb., MA, 2001.

[11] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. *Proc. IEEE International Symp. Parallel and Distributed Processing*, pages 349–359, 2014.

[12] T. Davis. The University of Florida Sparse Matrix Collection. *http://www.cise.ufl.edu/research/sparse/matrices/*.

[13] R. K. Ghosh and G. P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24:134–150, 1984.

[14] T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19:678–704, 1990.

[15] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal ACM.*

[16] J. JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[17] E. Korach and Z. Ostfeld. Recognition of DFS trees: Sequential and parallel algorithms with refined verifications. *Discrete Mathematics*, 114:305–327, 1993.

[18] J. Ma, K. Iwama, T. Takaoka, and Q. Gu. Efficient parallel and distributed topological sort algorithms. *Proc. International Symp. Parallel Algorithms and Architecture Synthesis*, page 378, 1997.

[19] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2012.

[20] C. Peng, B. Wang, and J. Wang. Recognizing unordered depth-first trees of an undirected graph in parallel. *IEEE Trans. Parallel and Distributed Systems*, 11:559–570, 2000.

[21] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[22] C. A. Schevon and J. S. Vitter. A parallel algorithm for recognizing unordered depth-first search. *Information Processing Letters*, 28:105–110, 1988.

[23] G. Shannon. A linear-processor algorithm for depth-first search in planar graphs. *Purdue University*, CS-TR-737, 1988.

[24] J. R. Smith. Parallel algorithms for depth-first searches i. planar graphs. *SIAM Journal on Computing*, 15:814–830, 1986.

[25] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[26] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *Foundations of Computer Science*, pages 12–20, 1984.

[27] Y. H. Tsin. Finding lowest common ancestors in parallel. *IEEE Transactions on Computers*, 35:764–769, 1986.

[28] U. Vishkin and R. E. Tarjan. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14:862–874, 1985.

[29] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the GPU. *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2016.

[30] Y. Zhang. A note on parallel depth first search. *BIT*, 26:195–198, 1986.